Using Streaming SIMD Extensions to Evaluate a Hidden Markov Model with Viterbi Decoding

Version 2.1

01/99

Order Number: 243645-002

Table of Contents

Revision History

Revision	Revision History	Date
2.1	FCS revision	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*, Rabiner, Lawrence R., Proceedings of the IEEE, February 1989, Vol. 77, No. 2.

2. *Speech Recognition: No Longer a Dream but Still a Challenge*, Quinnel, Richard A., EDN, January 19, 1995, pgs 41 – 46.

3. *Using MMX™ Instructions to Implement Viterbi Decoding*, Intel Application Note, AP-569, Copyright 1996, http://developer.intel.com/drg/mmx/appnotes/ap569.htm.

4. *Dynamic Programming*, Ch 7, pgs 7-1 to 7-24, Intel's Recognition Primitives Library.

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide floating point single-instruction, multiple-data (SIMD) instructions and additional SIMD integer instructions. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and 3D audio. This application note describes how the Streaming SIMD Extensions can be used to implement the Viterbi decoding algorithm to evaluate Hidden Markov Models. This application note presents two versions of the Viterbi algorithm, one implemented in SIMD floating point (32-bit data), and the other in SIMD short (16-bit data).

A previous application note, AP-569 titled *Using MMX™ Instructions to Implement Viterbi Decoding*, described how the MMX™ instructions could be used to gain a 2x improvement over scalar code. This gain was primarily due to using the MMX instructions to operate on two 32-bit data elements at a time. This application note shows that implementing the Streaming SIMD Extensions and operating on four data elements at a time (increasing the SIMD width by two) can further increase the performance gain.

2 Viterbi Decoding Algorithm

This section briefly discusses how the Viterbi decoding algorithm is used to evaluate a Hidden Markov Model (HMM). The reader is encouraged to use the references listed on page iv to study these topics in more detail. The reader may review Section 2.2 first, this section provides some background information on HMMs.

An HMM (λ) consists of a Markov chain of *n* states, numbered j = 1, 2, … , n [4]. Transitions can occur from one state *i* to another state *j*. An HMM is described by the parameter set {A, B, π} where:

- A is the matrix $\{a_{ij}\}$ called the transition probability, where a_{ij} is the probability of transition from state *i* to state *j*.
- B is the matrix $\{b_i(k)\}$ called the output probability, where $b_i(k)$ is the probability of producing observation *k* when in state *i*.
- π is the vector $\{\pi_i\}$ called the initial probability, where π_i is the probability of starting in state *i*.

Different HMM structures can be used depending on the types of transitions that can occur. This document only considers the constrained-jump model. As shown in Figure 1, this model limits the state transitions. Transitions can only occur from state *i* to state *i*, *i+1*, or *i+2* ($a_{ij} = 0$ for $j > i+2$ and $j<i$).

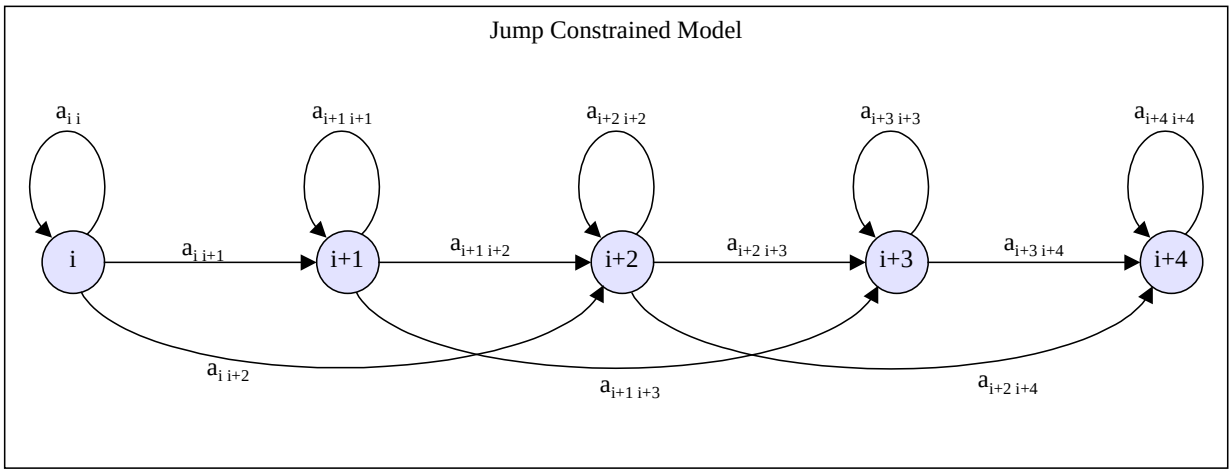


Figure 1: Jump Constrained Hidden Markov Model. Transitions can only occur from state i to state i, i+1, or i+2.

In this application note only discrete HMMs are considered. In a discrete HMM, there are M distinct observation symbols a state can generate. Each observation symbol corresponds to a particular physical output of the system being modeled. The individual observation symbols are denoted as $V = \{v_1, v_2, \ldots, v_M\}$. The indices of the observation symbols describe a sequence of physical outputs by means of an observation vector:

$O = \{O_1, O_2, \ldots, O_T\}$, where T is the number of observations in the sequence

Each individual observation in the observation vector is an index of an observation symbol that best matches the system's output. In a discrete HMM, the output probability $b_i(k)$ is the probability that the observation symbol v_k can be generated when in state *i* [1] (see Figure 2):

$$b_i(k) = P(v_k \text{ at time } t | q_t = \text{state } i), \text{ where } k = O_t,$$

$$1 \le i \le N,$$

$$1 \le k \le M$$

Given a discrete HMM with five states, each state has three probabilities associated to it:

π - Initial Probability
A - Transition Probability
B - Output Probability

Assuming this model has three observation symbols and is a jump constrained model, the probability matrices are as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ 0 & a_{22} & a_{23} & a_{24} & 0 \\ 0 & 0 & a_{33} & a_{34} & a_{35} \\ 0 & 0 & 0 & a_{44} & a_{45} \\ 0 & 0 & 0 & 0 & a_{55} \end{bmatrix} \quad B = \begin{bmatrix} b_1(1) & b_2(1) & b_3(1) & b_4(1) & b_5(1) \\ b_1(2) & b_2(2) & b_3(2) & b_4(2) & b_5(2) \\ b_1(3) & b_2(3) & b_3(3) & b_4(3) & b_5(3) \end{bmatrix} \quad \pi = \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \\ \pi_4 \\ \pi_5 \end{bmatrix}$$

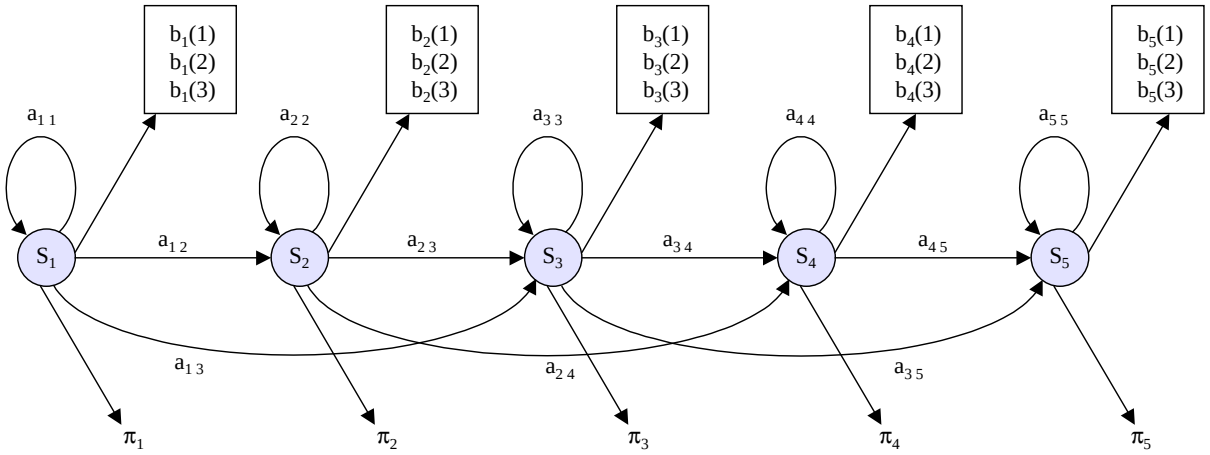


Figure 2: The parameters of a discrete Hidden Markov Model

HMMs can be used in speech recognition. A sequence of sound spectra generated by an utterance (words or syllables) can be modeled through a set of state movements [2]. Given a speech sample represented by an observation vector O, one can compute the probability that the observation sequence can be generated by a given HMM, $P(O|\lambda)$. As discussed in Section 2.2, the model may generate the same observation sequence through different state sequences, but each state sequence is associated with a different probability. The state sequence with the best probability is the path that is most likely to have generated the observation sequence. This state sequence is known as the best path (P^V):

$$P^V = \max[P(Q|O,\lambda)], \text{ where Q is a state sequence that can generate O}$$
$$P(O|\lambda) \approx P^V$$

To do an exhaustive search on an HMM for the best path is impractical (see Example 1). Instead, the Viterbi decoding algorithm can provide an efficient algorithm to find the best path. This algorithm can be used to find both the best path and the state sequence of the best path. For a fair comparison with application note AP-569, this implementation only solves for the best path and not the state sequences associated with the best path. That is, this implementation does not store the backtrace.

This example was taken from [1], pg 262.

Problem:

Find the probability of the observation sequence O, given the model λ, that is, find $P(O|\lambda)$. The observation sequence $O = \{O_1, O_2, \ldots, O_T\}$, where T is the number of observations in the sequence.

Background Information:

- Time instants associated with state changes are denoted as t = 1, 2, ...
- A state at time t is denoted as q_t.
- A state sequence of length T (the number of observations) is denoted as

  $Q = q_1, q_2, \ldots, q_T$, where q_1 is the initial state.

- The probability of the current state generating the observation at time t, given the model λ, is equal to the output probability $b_{q_t}(O_t)$:

  $$P(O_t|q_t, \lambda) = b_{q_t}(O_t)$$

- Assuming statistical independence of observations, the probability of generating the observation sequence giving a state sequence is equal to the product of the individual conditional probabilities.

  $$P(O|Q, \lambda) = \prod_{t=1}^{T} P(O_t|q_t, \lambda)$$

  $$= \prod_{t=1}^{T} b_{q_t}(O_t)$$

- The probability of the state sequence Q giving the model is equal to the product of the initial probability and the appropriate transition probabilities:

  $$P(Q|\lambda) = \pi_{q_1} \; a_{q_1 q_2} \, a_{q_2 q_3} \; \ldots \; a_{q_{T-1} q_T}$$

Solution:

- The probability that O and Q occur simultaneously (the joint probability), is the product of the two probabilities:

  $$P(O,Q|\lambda) = P(O|Q, \lambda) \, P(Q|\lambda)$$

- The probability of O given the model is equal to summing the joint probability over **ALL** possible state sequences Q ($2 * T * N^T$ calculations needed if any state can transition to any other state):

  $$P(O|\lambda) = \sum_{\text{All possible Q}} P(O|Q, \lambda) \, P(Q|\lambda)$$

  $$= \sum_{\text{All possible Q}} \pi_{q_1} b_{q_1}(O_1) \, a_{q_1 q_2} b_{q_2}(O_2) \, a_{q_2 q_3} b_{q_3}(O_3) \quad \ldots \quad a_{q_T q_{T-1}} b_{q_T}(O_T)$$

Example 1: How to find the probability of the observation sequence given the model

The Viterbi algorithm is used to find the best path (P^V) given an observation sequence O and a HMM model λ. The description of the Viterbi algorithm [1]:

- Define $\delta_t(i)$ to be the best score (highest probability) along a single path at time *t*. This score considers the first t observations and the state sequence ends in state S_i:

$$\delta_t(i) = \max_{q_1, q_2, ..., q_{t-1}} P(q_1 q_2 ... q_t = S_i, O_1 O_2 ... O_t | \lambda)$$

And

$$\delta_{t+1}(j) = [\max_i \delta_t(i) a_{ij}] * b_j(O_{t+1})$$

- Viterbi Algorithm:

1) Initialization:

$$\delta_1(i) = \pi_i b_i(O_1) \qquad 1 \le i \le N \qquad (1)$$

2) Recursion:

$$\delta_t(j) = \max_{i = j, j-1, j-2} [\delta_{t-1}(i) a_{ij}] * b_j(O_t) \qquad 2 \le t \le T, \; 1 \le j \le N \qquad (2)$$

3) Termination:

$$P^V = \max_{1 \le i \le N} [\delta_T(i)] \qquad (3)$$

2.1 Applications for Evaluating HMMs using the Viterbi Decoding Algorithm

The following discussion was taken from [1] pgs 276-277. One application for the Hidden Markov Model is in an Isolated Word Recognizer. Rabiner discusses how each HMM is designed and trained for each word *W* in the vocabulary of the system. The Isolated Word Recognizer first performs feature analysis on an incoming speech signal – a spoken word (see Figure 3). The feature analysis converts a speech signal into a time sequence of coded spectral vectors (the observation sequence). As discussed above, each observation is an index of an observation symbol that best matches the physical output. In this speech recognition system, the observation symbols are coded spectral vectors. Once the observation sequence is derived from the input signal, the probability of generating the observation sequence given an HMM is computed for all models, $P(O|\lambda)$. To find $P(O|\lambda)$, each HMM is evaluated by the Viterbi algorithm to find the best path per HMM. The best path represents how well the most likely sequence of states in an HMM explains the given sequence of observations. The spoken word is considered recognized when the model with best probability is found:

$$w^* = \underset{1 \le w \le W}{\text{argmax}} [P(O|\lambda^W)]$$

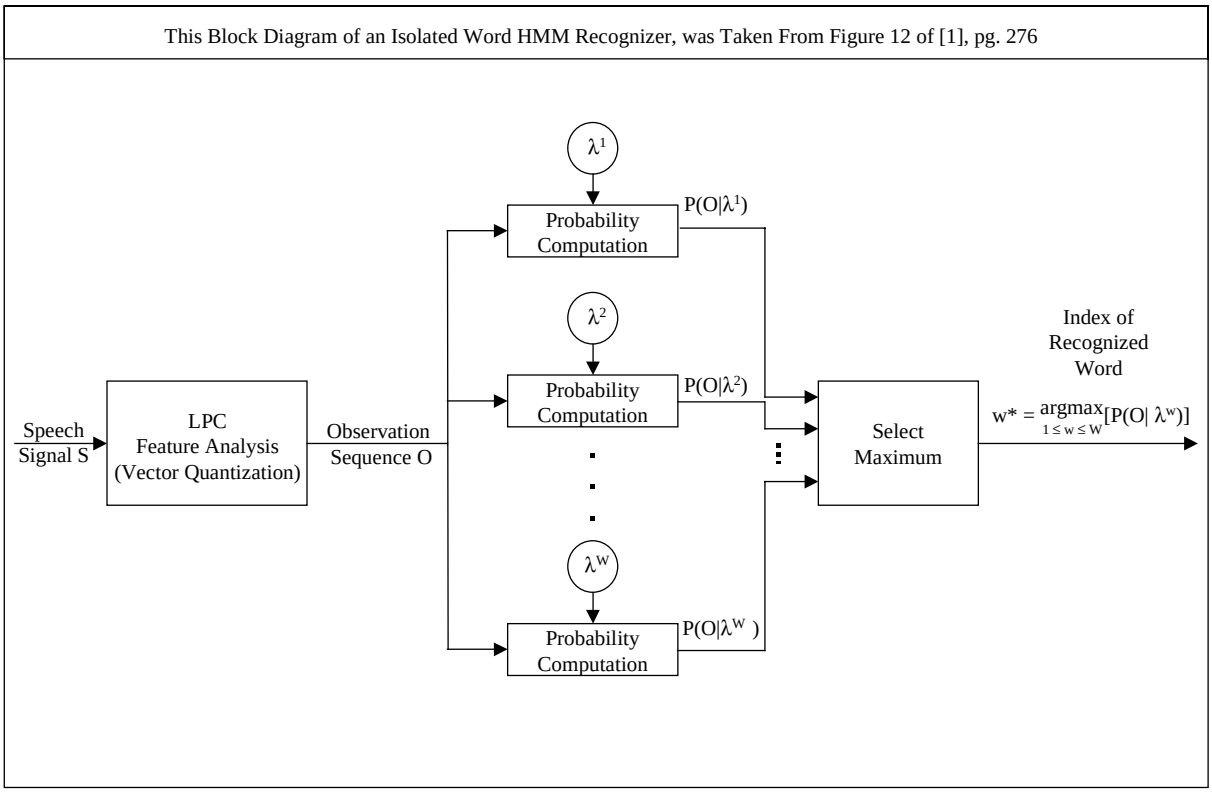


Figure 3: Block Diagram of Isolated Word Recognizer

2.2 Background Information on the Hidden Markov Model

The following example briefly explains the concepts of a hidden Markov model and how it differs from an observable Markov model. Suppose there are two experimenters E1 and E2 and two observers B1 and B2. Experimenter E1 throws a pair of dice and adds their sum. After a series of dice throws experimenter E1 produces a sequence of results or observations:

$$O = \{O_1, O_2, O_3, \ldots, O_4\}$$

O is the observation vector that stores the individual observations in the order of occurrence. Each individual observation corresponds to a sum of the pair of dice after each throw.

Experimenter E1 allows observer B1 to witness the following:

- The individual observations correspond to a sum of one pair of dice.
- The value of each die per dice throw.
- The resulting observation sequence.

With this information, observer B1 can design an observable Markov model to explain the observation sequence produced by experimenter E1. The states of the Markov model are chosen to represent the sum of the dice (see Figure 4). Each state can generate only one output. For example, state 1 can produce only the summation of 1+1 or 2 (Output probability is $b_1(2) = 1$). Observer B1 can use this model to rewrite the observation sequence in terms of the corresponding state sequence (see Figure 5).

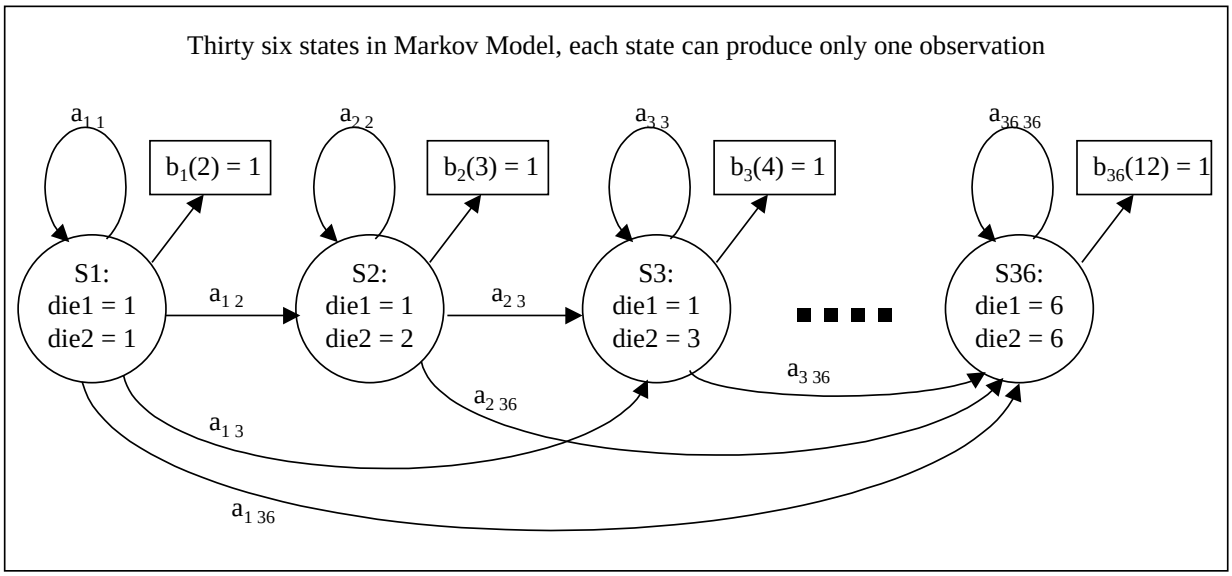


Figure 4: Markov model used by observer B1 and B2 to explain dice throw

Observation sequence produced by experimenter E1: O = {1+2 = 3, 3+4=7, 6+6=12, ..., 4+3=7}	Observation sequence produced by experimenter E2: O = {3, 7, 12, ..., 7}
Results as observed by observer B1: die1 = 1, die2 = 2, sum = 3 die1 = 3, die3 = 4, sum = 7 die1 = 6, die2 = 6, sum = 12 : : die1 = 4, die2 = 3, sum = 7	Results as observed by observer B2: Observer B2 does not know the value per die per given observation
Observation sequence in terms of corresponding state sequence of observable Markov Model: O = {S2, S16, S36, ... S21}	Cannot write observation sequence in terms of corresponding state sequence of hidden Markov Model * More than one state sequence corresponds to observation sequence.

Figure 5: Observer B1 can rewrite the observation sequence in terms of the corresponding state sequence, Observer B2 cannot rewrite the observation sequence

Like experimenter E1, experimenter E2 also has a pair of dice and produces an observation sequence that describes the sum of the dice per dice throw. Experimenter E2 allows observer B2 to witness the following:

- The individual observations correspond to a sum of one pair of dice.
- The resulting observation sequence

Unlike observer B1, observer B2 is not allowed to see the value per die after each dice throw. Observer B2 is only allowed to witness the observation sequence (the sum of the dice per dice throw). With this information, observer B2 can design a hidden Markov model to explain the observation sequence produced by experimenter E2. To explain the observation sequence, observer B2 decides to use the same model designed by observer B1 (see Figure 4). But, because different states can produce the same

output, observer B2 cannot rewrite the observation sequence in terms of the corresponding state sequence. Observer B2 does not have enough information to know which state produced a given observation (see Figure 5).

The model described in Figure 4 is used as an observable Markov model by observer B1, while observer B2 uses the same model as a hidden Markov model. An observable Markov model is a Markov model where the states have a one-to-one correspondence with the observations. In the observable Markov model, the observer is able to rewrite the observation sequence with the corresponding state sequence. In the hidden Markov model there can be more than one sequence of states that can produce the observation sequence.

2.3 Implementing the Viterbi Decoding Algorithm

The floating point implementation of the Viterbi algorithm only works if all vectors and arrays of pointers to arrays are aligned on a 16-byte boundary.

The application note AP-569 implemented the Viterbi decoding algorithm with the MMX instructions. For a fair comparison, the algorithm and notations used in the previous application note are also used here.

The three equations that make up the Viterbi algorithm discussed in Section 2, are implemented by taking the negative logarithms on each side. This is done to convert all multiplication operations into additions of positive numbers, in an attempt to avoid scaling issues. Using the previous application note's notation, the three equations of the Viterbi Algorithm are:

1) Initialization:

$$Dist(j,1) = Pi(j) + bProb(j,k(1)) \qquad 1 \le j \le N \qquad (4)$$

2) Recursion:

$$Dist(j,t+1) = \min_{i = j,\, j-1,\, j-2} [Dist(i,t) + aProb(j,i)] + bProb(j,k(t+1)) \qquad 1 \le t \le T,\ 1 \le j \le N \qquad (5)$$

3) Termination:

$$Dist^v = \min_{1 \le j \le N} Dist(j,T) \qquad (6)$$

The author of the previous application note uses the following notation:

$Dist^v$	$= -\log(P^V)$
$Dist(j,t)$	$= -\log(\delta_t(j))$
$aProb(j,i)$	$= -\log(a_{ij})$
$bProb(j,k(t))$	$= -\log(b_j(O_t))$
$Pi(j)$	$= -\log(\pi_j)$

Equation 5 is where most of the processing time of this algorithm is spent. This equation can be expanded as follows:

$$\text{Dist}(j,t+1) = \min\left[\{\text{Dist}(j,t) + \text{aProb}(j,j)\}, \{\text{Dist}(j-1,t) + \text{aProb}(j,j-1)\}, \{\text{Dist}(j-2,t) + \text{aProb}(j,j-2)\}\right] + \text{bProb}(j,k(t+1)) \quad \begin{matrix} 1 \le t \le T \\ 1 \le j \le N \end{matrix} \qquad (7)$$

This equation can be placed in SIMD form as demonstrated in Figure 6:

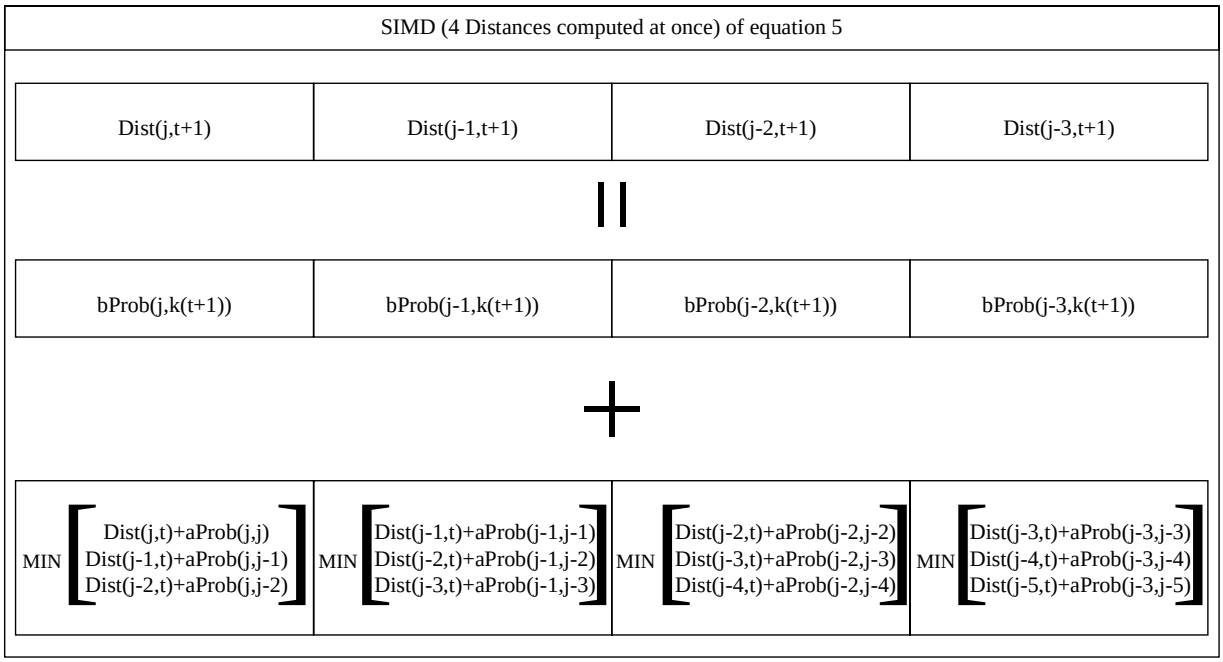


Figure 6: Equation 5 in SIMD form. Notice the left-most column is the scalar version of Eq 5.

In order to use the SIMD form of equation 5, the distances are computed by using decreasing values of j, (from j = N, N-1, ... , 1). The SIMD form of this equation is the core loop in both the floating point and short implementations of this algorithm. The core loop calculates the distances of four states at a time. After exiting the core loop, the distances of any remaining states (0, 1, 2, or 3 states) are computed.

For example, in the short implementation, the core loop calculates the distances of four states at a time through the following steps:

1. Load the next distances in three SIMD registers:

mm0 = [d3	d2	d1	d0	]
mm1 = [d4	d3	d2	d1	]
mm2 = [d5	d4	d3	d2	]

Where the notation d0 = Dist(j-0,t), d1 = Dist(j-1,t), …, d5 = Dist(j-5,t).

2. Add corresponding state transition probabilities:

```
mm0 = [d3+a(j-3,j-3)  d2+a(j-2,j-2)  d1+a(j-1,j-1)  d0+a(j-0,j-0)  ]
mm1 = [d4+a(j-3,j-4)  d3+a(j-2,j-3)  d2+a(j-1,j-2)  d1+a(j-0,j-1)  ]
mm2 = [d5+a(j-3,j-5)  d4+a(j-2,j-4)  d3+a(j-1,j-3)  d2+a(j-0,j-2)  ]
```

   Where the letter a is used to represent aProb. If the state transition probability matrix, aProb, is implemented by storing the probabilities in one vector as discussed in Section 3.2, the following notation can be used:

```
mm0 = [d3+a3     d2+a2     d1+a1     d0+a0     ]
mm1 = [d4+a7     d3+a6     d2+a5     d1+a4     ]
mm2 = [d5+a11    d4+a10    d3+a9     d2+a8     ]
```

   Where the vetor elements are as follows: a0 = a(j-0,j-0), a1 = (j-1,j-1), …, a4 = a(j-0,j-1), …, a11 = (j-3,j-5)

3. Find the minimum sum:

```
mm0 = [min3      min2      min1      min0      ]
```

   Where min0 = min(d0+a0,d1+a4,d2+a8), …, min3 = min(d3+a3,d4+a7,d5+a11)

4. Add the corresponding output probabilities, bProb (see Figure 6).

5. Store the calculated distances.

Even though Step 2 of the Viterbi Algorithm is where most of the processing time is spent, Steps 1 and 3 of the algorithm also can be optimized through the use of SIMD instructions. In Step 1, four distance initializations of a discrete HMM can be processed simultaneously. The initializations of distances of any remaining states are computed after the SIMD initialization loop.

Similarly, finding the minimum distance, as described in Step 3, also can be processed with SIMD (if there are eight or more distances). The following steps describe the floating point implementation of Step 3 (the short implementation is similar but it does not require the loads to be aligned):

1. Load the first four distances into the first SIMD floating point register (assume the vector is ALIGNED). Assume these four elements are the minimum four distances (see Figure 7).

2. Loop to get the minimum four aligned distances:

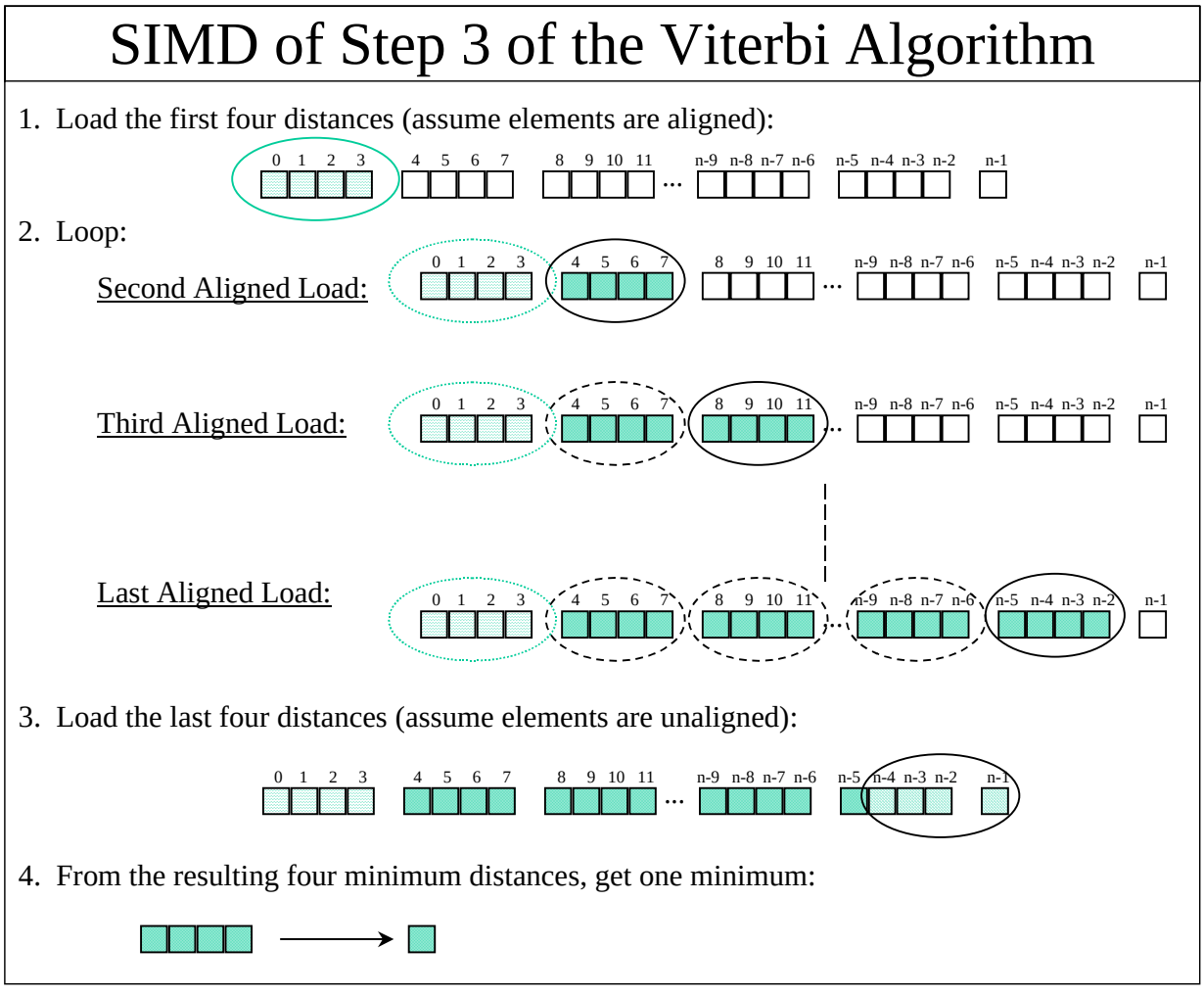
   - Load the next four ALIGNED elements into a second SIMD floating point register.
   - Compare these current four elements (second register) with the minimum four distances (first register). Store the resulting minimum four elements as the new minimum four distances (first register).

   Repeat the loop by loading the next four ALIGNED elements for the next comparison. The first register is used to hold the four minimum distances of the vector. Each of the individual minimum elements in the first register is the minimum of all elements that have occupied the corresponding

register field of both the first and second registers. The loop ends when either of the following conditions is met:

- Only the remaining elements (elements not part of the last aligned group of four elements) need to be loaded and compared - the vector in Figure 7 has one element remaining.
- If the vector has no remaining elements, end the loop when the last four aligned elements remain to be loaded and compared.

3. After the loop has completed, load the last four elements of the vector into the second SIMD register. Assume the last four elements are UNALIGNED. If the elements at the end of the array are unaligned, the last four elements overlap with the previous load (see Step 3 in Figure 7). Compare these last four elements (second register) with the four minimum distances (first register), and if necessary, update the first register to hold the new minimum four distances.
4. From the four minimum distances, shuffle and compare the four minimum elements to find one minimum distance. Return this minimum distance.



1?

Figure 7: Pictorial description of the steps used to find the minimum distance in the floating point implementation. Notice the overlapping of the loads in Step 3.

2.3.1 Techniques

The functions of the implementations discussed in this application note and in the application note AP-569 take as input seven arguments in the following order:

1.	obsVect	This is the observation vector that describes a speech signal as a sequence of observations, where each observation is an index that corresponds to an observation symbol.
2.	obsLen	The length of the observation sequence = T.
3.	aProb	The transition probability matrix that can be represented as a vector. The location and data type of the aProb elements depend on which implementation is being used (see Section 3.2).
4.	bProb	The output probability is an array of pointers to arrays. The size of the array holding the pointers is also the number of observation symbols, M. The size of the array pointed by each pointer is the number of states + 3 (see Figure 8). The three extra elements are for padding (see Section 3.2). The data type of the elements in bProb depends on which implementation is being used.
5.	Pi	The initial probability vector. The data type of the elements in this vector depends on the implementation used.
6.	nStates	The number of states in the HMM.
7.	distBuffer	The temporary buffer that stores the accumulated distance quantities.

To prevent overflow, the author of the previous application note used 32-bit data elements (integers/double words) to calculate and store the distances. For most applications this is an unnecessary precaution. Instead, the 16-bit data elements (short/word) can be used along with the pack add with saturate instruction (`paddsw`) to prevent overflow. The floating point SIMD implementation (23 precision bits per floating point number) also can be used without any overflow problems.

As discussed above, the output probability, bProb, is implemented as an array of pointers to arrays (see Figure 8). This implementation of the output probability assumes discrete HMMs (see Section 2).

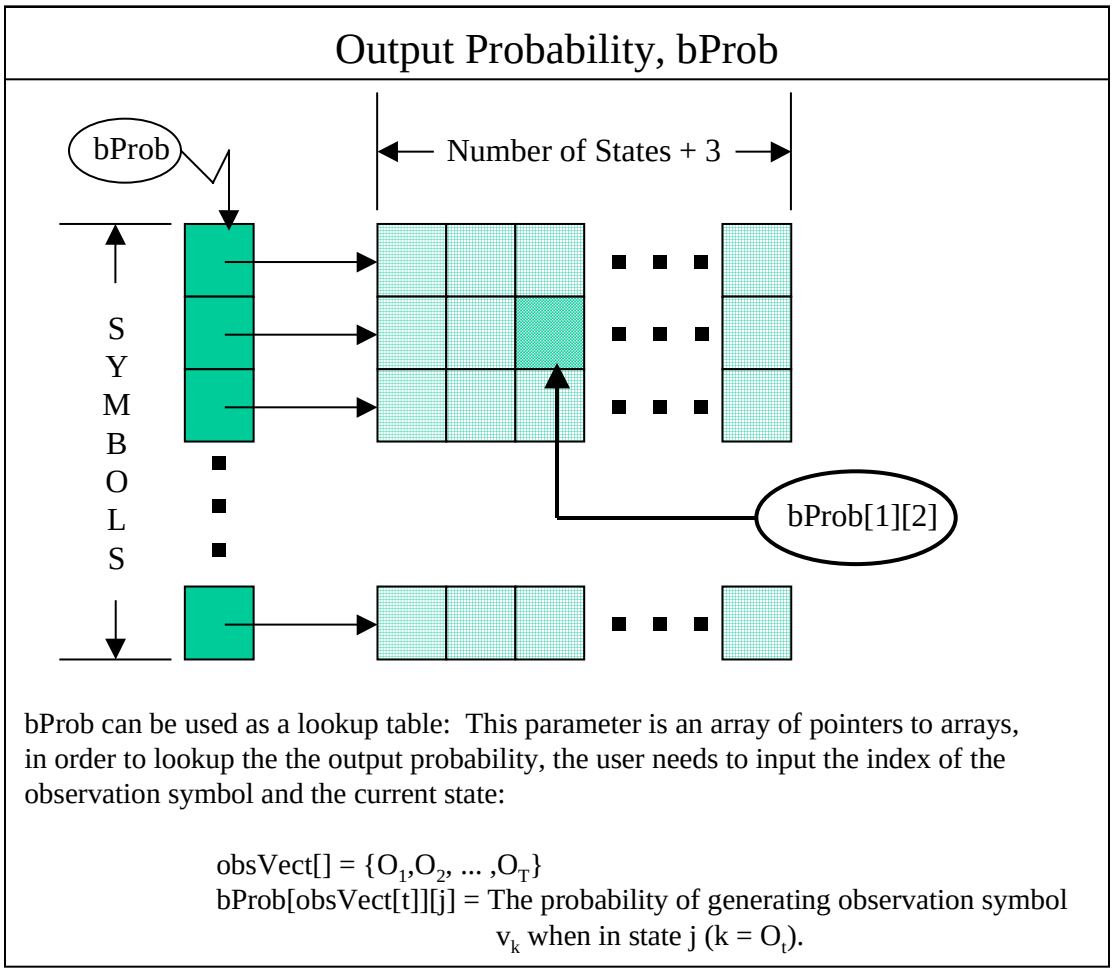


Figure 8: Pictorial description of how the output probability (bProb) is implemented

3 Performance

Increased SIMD width (processing four data elements at a time instead of two data elements at a time) employed in the short implementation may be the largest factor producingspeedup. Additional gain can be attributed to three Streaming SIMD Extensions: `pminsw`, `pinsrw`, and `pextrw`. In AP-569, the author described how five MMX instructions could be used to find the minimum of two values. In the short implementation, these five instructions can be replaced by the `pminsw` instruction. As discussed in Section 2.3.1, the short implementation could be used only if the saturated additions were used to prevent overflow. The `pinsrw` and `pextrw` instructions were used along with the `paddsw` instruction to provide saturated addition on scalar quantities.

Unlike the short implementation, the floating point SIMD instructions have more latency and less throughput than the integer SIMD instructions. This means that the increased latency and decreased throughput offsets the greater SIMD width provided by the floating point version of the Streaming SIMD Extensions. For this reason, most of the speedup is due to the `minps` instruction that allowed five instructions used in AP-569 to be reduced to one instruction.

Readers should be aware that although the performance gain for the floating point implementation is modest,the floating point implementation allows for a greater dynamic range than the MMX implementation. Also, this paper did not discuss the prefetch instruction that is part of the Streaming

SIMD Extensions. Additional gains may be seen by using the prefetch instruction, because the data access patterns for this algorithm are very regular.

3.2 Considerations

As stated above, the transition matrix can be turned into one vector, aProb. In scalar code (no SIMD used) the aProb vector can be described as follows:

aProb[] = {..., a(j,j), a(j,j-1), a(j,j-2), a(j-1,j-1), a(j-1,j-2), a(j-1,j-3), a(j-2,j-2),... }
where a(j,j-1) means the transition probability from state j-1 to state j.

Unfortunately, this vector has to be modified for the different SIMD versions. Application Note AP-569 describes how the aProb vector is modified for a SIMD width of two (two data elements per instruction) process. This implementation uses a SIMD width of four as discussed in section 2. To use the SIMD width of four, the aProb vector must be as follows:

aProb[] = {..., a(j,j), a(j-1,j-1), a(j-2,j-2), a(j-3,j-3), a(j,j-1), a(j-1,j-2), a(j-2,j-3), a(j-3,j-4), a(j, j-2), a(j-1,j-3), a(j-2,j-4), a(j-3,j-5), a(j-4,j-4), a(j-5,j-5), a(j-6,j-6), a(j-7,j-7), a(j-4,j-5),....}

Proper padding is another important consideration. When the last four distances (j = 3, j-1 =2, j-2=1, j-3 =0) are being computed in a model that has a multiple of four states, the core loop will assume that there are two more states (j-4 and j-5). To use the core loop effectively, the user needs to pad the last two distance values in the distance vector with high values (Dist(j-4,t) = HI, Dist(j-5,t) = HI, where HI represents a large number). In this implementation the distance values are stored in one vector, `distBuffer`. To pad the last two distance values with high values means `distBuffer[nStates] = HI`, and `distBuffer[nStates+1] = HI`, where `nStates` is the number of states in the model. Also, when computing the distances of the remaining states (the states that are not part of the last aligned group of four states), special care needs to be taken so the necessary aProb and bProb values are also padded. The following code can be used to pad the appropriate aProb elements with high values:

```
////////////////////////////////////////////////////
// Place MaxShort and Infinity values in proper place
////////////////////////////////////////////////////
// assume mnStates, maProb_fp, and maProb_sh are defined above

//Max Elements:  f_infinity, MaxShort
int   const i_infinity = 0x7F800000;
float const f_infinity = *(float*)&i_infinity;
short const MaxShort   = 0x7FFF;

   int aPlen = 0;   // number of aProbs

   //Get aProb length
   if (mnStates > 3)
   {
      // There are 12 aProbs elements for every 4 States.
      aPlen = (mnStates &~ (0x03)) * 3;
   }
```

```
  if ((mnStates & (0x03))> 0)
  {
    // There are 4 aProbs elements for every remaining State
    aPlen += ((mnStates & (0x03))*4);
  }

////////////////////////////////////////////////////////////////////////////
  assert(aPlen>=(mnStates*3));

  maProb_fp[aPlen-1] = f_infinity;
  maProb_fp[aPlen-2] = f_infinity;

  maProb_sh[aPlen-1] = MaxShort;
  maProb_sh[aPlen-2] = MaxShort;

  switch(mnStates & (0x03))
  {
    // multiple of four states, no extra or remaining states
    case 0:
    {
      maProb_sh[aPlen-5] = MaxShort;
      maProb_fp[aPlen-5] = f_infinity;
      break;
    }

    // one extra state
    case 1:
    {
      maProb_fp[aPlen-3] = f_infinity;
      maProb_sh[aPlen-3] = MaxShort;
      if (mnStates>4) {
        maProb_sh[aPlen-5] = MaxShort;
        maProb_fp[aPlen-5] = f_infinity;
      }
      break;
    }

    // two extra states
    case 2:
    {
      maProb_sh[aPlen-3] = MaxShort;
      maProb_sh[aPlen-5] = MaxShort;
```

```
        maProb_sh[aPlen-6] = MaxShort;

        maProb_fp[aPlen-3] = f_infinity;
        maProb_fp[aPlen-5] = f_infinity;
        maProb_fp[aPlen-6] = f_infinity;
        break;
      }

      // three extra states
      case 3:
      {
        maProb_sh[aPlen-3] = MaxShort;
        maProb_sh[aPlen-5] = MaxShort;
        maProb_sh[aPlen-6] = MaxShort;
        maProb_sh[aPlen-9] = MaxShort;

        maProb_fp[aPlen-3] = f_infinity;

        maProb_fp[aPlen-5] = f_infinity;
        maProb_fp[aPlen-6] = f_infinity;
        maProb_fp[aPlen-9] = f_infinity;
        break;
      }
    }
```

Similarly, the appropriate bProb elements must also be padded. As discussed in Section 2.3.1, the size of the array pointed by each pointer is the number of states + 3. The three extra elements are used for padding. The following code can be used to pad the three extra bProb elements with high values:

```
    for (j = 0; j < msymbols; j++)
    {
      mbProb_sh[j][mnStates]   = MaxShort;
      mbProb_sh[j][mnStates+1] = MaxShort;
      mbProb_sh[j][mnStates+2] = MaxShort;

      mbProb_fp[j][mnStates]   = f_infinity;
      mbProb_fp[j][mnStates+1] = f_infinity;
      mbProb_fp[j][mnStates+2] = f_infinity;
    }
```

4 Conclusion

Streaming SIMD Extensions can provide a significant performance gain in evaluating HMMs through the with the Viterbi algorithm. The `pminsw`, `pinsrw`, and `pextrw` Streaming SIMD Extensions

contributed to this speedup. The floating point implementation also provided a speedup , mainly due to the `minps` instruction.

5 C Coding Example

The following C code was taken from Application Note AP-569. Performance Analysis was done on this C code and the results are listed in section 3.

```
#include <stdio.h>
#include <stdlib.h>


/* Return the minimum of two integers. Note the macro __min can be used
 * directly if attention is paid to the parenthesis in the arithmetic
 * expressions.
 */
int imin(int a, int b)
{
        int i;

        i = __min(a, b);

        return i;
} /* imin */

/* C implementation for the Viterbi function for discrete HMMs. The arguments are
 * described briefly here but described in detail in the accompanying application notes.
 * obsVect is the vector of VQ indices corresponding to the continuous observations
 * obsLen is the length of the observation
 * aProb is the vector of transition probabilities (format described in the appliation notes)
 * bProb is the matrix of output probabilities. bProb points to a vector of pointers to row
 * vectors. Each row index corresponds to the VQ index and each column index corresponds to
 * the state number.
 * pi is the vector of initial probabilities.
 * nStates is the number of states.
 * distBuffer is used by the function for temporary storage of distances.
 */

int viterbi_c(unsigned int *obsVect, int obsLen, unsigned int *aProb,
        unsigned int **bProb, unsigned int *pi, int nStates, int *distBuffer)
{
        int i, j, minDist;
        int *Dist = distBuffer;
        unsigned int *b, *aProbTemp = aProb;
        int nCount, nRemain;
        short temp = 3;

      /* Initialiazation */
        for (i = 0; i < nStates; i++) {
                Dist[i] = pi[i] + bProb[obsVect[0]][i];
        }
```

```
/* nCount will contain the number of times to execute the main loop
 * which processes four states at a time. nRemain contains the number
 * of states remaining after processing four at a time (i.e. 0, 1, 2 or 3)
 */
nCount = nStates >> 2;
nRemain = nStates - (nCount << 2);

/* Iterate through each observation (i.e. increment time index) */
for (i = 1; i < obsLen; i++)
{
        b = bProb[obsVect[i]];
        Dist = distBuffer;
        aProb = aProbTemp;

        /* Process four states at a time */
        for (j = 0; j < nCount; j++) {
                Dist[0] = b[0] + imin(Dist[0] + aProb[0], imin(Dist[1] + aProb[2],
                        Dist[2] + aProb[4]));
                Dist[1] = b[1] + imin(Dist[1] + aProb[1], imin(Dist[2] + aProb[3],
                        Dist[3] + aProb[5]));
                Dist[2] = b[2] + imin(Dist[2] + aProb[6], imin(Dist[3] + aProb[8],
                        Dist[4] + aProb[10]));
                Dist[3] = b[3] + imin(Dist[3] + aProb[7], imin(Dist[4] + aProb[9],
                        Dist[5] + aProb[11]));

                /* Update addresses for the nex four states */
                Dist = Dist + 4;
                b = b + 4;
                aProb = aProb + 12;
        } /* for */

        /* Process the remaining states if any (1, 2 or 3 remaining states) */
        switch (nRemain)
        {
                case 3 :
                        {
                        Dist[0] = b[0] + imin(Dist[0] +aProb[0], imin(Dist[1]+aProb[2],
                                Dist[2] + aProb[4]));
                        Dist[1] = b[1] + imin(Dist[1] + aProb[1], Dist[2] + aProb[3]);
                        Dist[2] = b[2] + Dist[2] + aProb[6];
                        break;
                        }
                case 2 :
                        {
                        Dist[0] = b[0] + imin(Dist[0] + aProb[0], Dist[1] + aProb[2]);
                        Dist[1] = b[1] + Dist[1] + aProb[1];
                        break;
                        }
                case 1 :
```

```
                                {
                                Dist[0] = b[0] + Dist[0] + aProb[0];
                                break;
                                }
                        } /* case */

        } /* for */

        /* Find and return the minimum distance in distBuffer */
        Dist = distBuffer;
        minDist = distBuffer[0];
        for (i = 1; i < nStates; i++)
                if (Dist[i] < minDist)
                        minDist = Dist[i];

        return minDist;
} /* viterbi_c */
```

6 Streaming SIMD Extensions Assembly Code Example

Two implementations using the Streaming SIMD Extensions were developed: Floating point implementation and Short implementation.

6.1 Floating Point Implementation, Assembly

```
#include <stdio.h>
#include <stdlib.h>

static const int Infinity = 0x07F800000;
static const int mask0[4] = {0,        0,     0,              0       };
static const int mask1[4] = {0,        Infinity, Infinity, Infinity};


float viterbi_fp(unsigned int *obsVect, int obsLen, float *aProb,
                 float **bProb, float *pi, int nStates, float *distBuffer)
{

   unsigned int set4States       = nStates & ~(0x03);
   unsigned int set4States_x_4   = set4States << 2;
   unsigned int nRemain          = nStates - set4States;
   float     *lastFour           = (distBuffer+nStates-5);
   float     minDist;
   unsigned int SIMD4MinCount  = ((nStates-1) &~ (0x03))<<2;

   _asm{
      // Initialize Distance
      ; Init registers to point to base addresses
      mov   esi, bProb        ; esi = &bProb[0]
      mov     eax, obsVect       ; eax = &obsVect[0]
      mov   eax, [eax]        ; eax = obsVect[0]
      mov   esi, [esi + eax*4]   ; esi = &bProb[obsVect[0]][0]
      mov   edi, distBuffer      ; edi = &distBuffer[0]
      mov   ebx, pi           ; ebx = &pi[0]

      mov   ecx, nRemain
      mov     edx, set4States_x_4   ; edx = set4States*4
      add  esi, edx
      add     ebx, edx
      add  edi, edx
      neg     edx

      cmp   edx, 0            ; if States are divided in sets of four
      jnz   Init4Dist            ;;; Call init4Dist to init w/ SIMD

      cmp   ecx, 0            ; else if States are not divided in sets of four
      jnz   Init1Dist            ;;; Call init1Dist to init w/o SIMD
```

```
      jmp   Exit              ; else Exit when HMM model contains zero states

      //Use SIMD to initialize Distance while states are in sets of four
Init4Dist:
      movaps xmm0, [ebx+edx]     ; xmm0    = [pi3     pi2     pi1     pi0    ]
      addps  xmm0, [esi+edx]     ; xmm0    = [pi3+b03 pi2+b02 pi1+b01 pi0+b00]
      movaps[edi+edx], xmm0      ; dist[] = xmm0
      add    edx, 16
      jnz    Init4Dist

      cmp     ecx, 0
      jz   InitDone

      // edx equals zero before entering loop
      //Initialize Distance of remaining states, states not in a set of four
Init1Dist:
      movss xmm0, [ebx+edx*4]    ; xmm0    = pi[i]
      addss xmm0, [esi+edx*4]    ; xmm0    = pi[i] + bProb[ObsVect[0]][i]
      movss [edi+edx*4],   xmm0  ; dist[i] = pi[i] + bProb[ObsVect[0]][i]
      inc     edx
      cmp  edx, ecx
      jne     Init1Dist

InitDone:
      movaps  xmm3, mask0     ; xmm3 = [ 000...0   000...0   000...0  000...0]
      movaps  xmm4, mask1     ; xmm4 = [+Infinity +Infinity +Infinity 000...0]
      cmpeqps xmm3, xmm4      ; xmm3 = [ 000...0   000...0   000...0  111...1]

      mov   edi, distBuffer      ; edi = &distBuffer[0]
      xor     ecx, ecx

      //Loop for each observation or obsCount
mainLoop:
      mov   edx, set4States_x_4
      inc   ecx
      cmp      ecx, obsLen
      je    mainLoopDone
      mov      esi, bProb           ; esi = &bProb[0]
      mov      eax, obsVect         ; eax = &obsVect[0]
      mov      eax, [eax+ecx*4]     ; eax = obsVect[ecx]
      mov      esi, [esi+eax*4]     ; esi = &bProb[obsVect[ecx]][0]
      mov      ebx, aProb
      xor   eax, eax

      //Loop to Find the Dist for each state per observation
      cmp    edx, 0
      je     CheckLastStates
```

```
FourStateLoop:
      movaps xmm0, [edi+eax]        ; xmm0 = [d3     d2     d1     d0   ]
      movups xmm1, [edi+eax+4]      ; xmm1 = [d4     d3     d2     d1   ]
      addps  xmm0, [ebx+eax]        ; xmm0 = [d3+a3  d2+a2  d1+a1  d0+a0]
      addps  xmm1, [ebx+eax+16]     ; xmm1 = [d4+a7  d3+a6  d2+a5  d1+a4]
      movups xmm2, [edi+eax+8]   ; xmm2 = [d5     d4     d3     d2   ]
      addps xmm2, [ebx+eax+32]   ; xmm2 = [d5+a11 d4+a10 d3+a9  d2+a8]
      minps xmm0, xmm1
      minps xmm0, xmm2           ; xmm0 = min(Dist+aProb) = min(xmm0,xmm1,xmm2)
      addps xmm0, [esi+eax]      ; xmm0 += bProb[obsVect[ecx]][i]
      movaps [edi+eax], xmm0     ; store distance
      add      eax,  16
      add      ebx,  32
      cmp      eax,  edx
      jnz      FourStateLoop

CheckLastStates:
      mov      edx, nRemain
      cmp       edx, 1        ; if one state left, handle last State
      je    lastState
      cmp      edx, 2            ; if two states left, handle the last two states
      je    lastTwoStates
      cmp      edx, 3            ; if three states left, handle the three states
      je    lastThreeStates
      jmp      mainLoop       ; else find distance for a new observation


lastThreeStates:
      ; Three states left to process

      movups   xmm2, [edi+eax+8]    ; xmm2 = [xx   xx    xx    d2   ]
      movups   xmm1, [edi+eax+4]    ; xmm1 = [xx   xx    d2    d1   ]
      addps    xmm2, [ebx+eax+32]   ; xmm2 = [xx   xx    xx    d2+a8]

      //Can use two shuffles instead of and, or combination
      andps xmm2, xmm3           ; xmm2 = [0    0     0     d2+a8]
      addps xmm1, [ebx+eax+16]   ; xmm1 = [xx   xx    d2+a5 d1+a4]
      movaps xmm0, [edi+eax]     ; xmm0 = [xx   d2    d1    d0   ]
      addps xmm0, [ebx+eax]      ; xmm0 = [xx   d2+a2 d1+a1 d0+a0]

      orps  xmm2, xmm4           ; xmm2 = [+Inf +Inf  +Inf  d2+a8]
      minps xmm1, xmm2           ; xmm1 = [xx   xx    d2+a5 min  ]
      shufps xmm1, xmm4,0x0F4    ; xmm1 = [+Inf +Inf  d2+a5 min  ]
      minps xmm0, xmm1           ; xmm0 = [xx   d2+a2 min1  min0 ]
      addps xmm0, [esi+eax]      ; xmm0 += bProb[obsVect[ecx]][i]
      movaps   [edi+eax], xmm0   ; store distance
      jmp     mainLoop

lastTwoStates:
      ; Two states left to process
```

```
      movss xmm1, [edi+eax+4]    ; xmm1 = [xx      d1   ]
      addss xmm1, [ebx+eax+16]   ; xmm1 = [xx      d1+a4]

      // Can use two shuffles instead of and, or combination
      andps xmm1, xmm3        ; xmm1 = [000...0 d1+a4]
      movlps   xmm0, [edi+eax]   ; xmm0 = [d1    d0   ]
      addps xmm0, [ebx+eax]      ; xmm0 = [d1+a1 d0+a0]
      orps  xmm1, xmm4           ; xmm1 = [+Inf  d1+a4]
      minps xmm0, xmm1           ; xmm0 = [d1+a1 min  ]
      addps xmm0, [esi+eax]      ; xmm0 += bProb[obsVect[ecx]][i]
      movlps   [edi+eax], xmm0   ; store distance
      jmp      mainLoop

lastState:
      ; Last state to process
      movss xmm0, [edi+eax]      ; xmm0 = [xx  xx  xx  d0   ]
      addss xmm0, [ebx+eax]      ; xmm0 = [xx  xx  xx  d0+a0]
      addss xmm0, [esi+eax]      ; xmm0 += bProb[obsVect[ecx]][i]
      movss    [edi+eax], xmm0   ; store distance
      jmp      mainLoop

      // Get Minimum Distance
mainLoopDone:
      mov   ecx, nStates
      cmp      ecx, 8
      jg    SIMD4          ; Use SIMD if enough states are used in HMM model

      // If Less than nine states find the minimum distance w/o employing SIMD
      movss xmm0, [edi]    ; xmm0 = [xx xx xx m0]
      cmp      ecx, 1
      je    storeMin
      shl      ecx, 2
      add      edi, ecx       ; edi = &dist[nStates-1]
      neg      ecx               ; [edi+ecx] holds next distance
loopMin:
      add      ecx, 4            ; increment pointer to load next distance
      movss xmm1, [edi+ecx]   ; load next distance
      minps xmm0, xmm1        ; place minimum distance in mm0
      jnz      loopMin

storeMin:
      movss minDist, xmm0     ; store minimum distance and exit
      jmp      Exit

      // If nStates > 4, use SIMD to find minimum distance
SIMD4:
      mov      ecx, SIMD4MinCount   ; ecx = num of states in sets of four
      add      edi, ecx          ; edi = &dist[SIMD4MinCount-1]
      neg      ecx                  ; [edi+ecx] will hold next four distances
```

```
      movaps   xmm0, [edi+ecx]      ; load first four distances
      add      ecx, 16           ; increment pointer to load next four distances

loopMinSIMD4:
      movaps   xmm1, [edi+ecx]      ; load next four distances
      minps xmm0, xmm1        ; place minimum four distances in mm0
      add      ecx, 16           ; increment pointer to load next four distances
      jnz   loopMinSIMD4

      // load the last four distances, store the minimum four distances
      mov       edi,  lastFour      ; edi = &dist[nstates-5]
      movups    xmm1, [edi]         ; load last four distances
      minps  xmm0, xmm1       ; place minimum four distances in mm0

      // Get the minimum distance from the four minimum distances
                              ; xmm0 = [m3   m2   m1   m0  ]
      shufps xmm5, xmm0, 0x40 ; xmm5 = [m1   m0   xx   xx  ]
      minps  xmm5, xmm0       ; xmm5 = [min2 min1 xx   xx  ]
      shufps xmm0, xmm5, 0x30 ; xmm0 = [xx   min2 xx   xx  ]
      minps  xmm5, xmm0       ; xmm5 = [xx   min  xx   xx  ]
      shufps xmm5, xmm5, 0xAA ; xmm5 = [min  min  min  min ]

      movss  minDist, xmm5    ; store min
      }
Exit:
   return(minDist);
  }
```

6.2 Short Implementation, Assembly

```
#include <stdio.h>
#include <stdlib.h>

static const short MaxShort = 0x7FFF;
static const short mask0[4] = {MaxShort,MaxShort,MaxShort,MaxShort};
static const short mask1[4] = {0,       MaxShort,MaxShort,MaxShort};
static const short mask2[4] = {0,       0,       MaxShort,MaxShort};
static const short mask3[4] = {0,       0,       0,       MaxShort};


short viterbi2_mmx(unsigned int *obsVect, int obsLen, short *aProb,
        short **bProb, short *pi, int nStates, short *distBuffer)
{

   unsigned int set4States     = nStates & ~(0x03);
   unsigned int set4States_x_2 = set4States << 1;
   unsigned int nRemain        = nStates - set4States;
   short     *lastFour         = (distBuffer+nStates-4);
   short     minDist;
   unsigned int SIMD4MinCount  = ((nStates-1) &~ (0x03))<<1;

   _asm{
      // Initialize Distance
      ; Init registers to point to base addresses
      mov   esi, bProb        ; esi = &bProb[0]
      mov      eax, obsVect         ; eax = &obsVect[0]
      mov   eax, [eax]        ; eax = obsVect[0]
      mov   esi, [esi + eax*4]   ; esi = &bProb[obsVect[0]][0]
      mov   edi, distBuffer      ; edi = &distBuffer[0]
      mov   ebx, pi           ; ebx = &pi[0]

      mov   edx, set4States
      mov   ecx, nRemain

      shl     edx, 1          ; edx = set4States*2
      add  esi, edx
      add     ebx, edx
      add  edi, edx
      neg     edx

      cmp   edx, 0            ; if States are divided in sets of four
      jnz   Init4Dist            ;;; Call init4Dist to init w/ SIMD

      cmp   ecx, 0            ; else if States are not divided in sets of four
      jnz   Init1Dist            ;;; Call init1Dist to init w/o SIMD

      jmp   Exit              ; else Exit when HMM model contains zero states
```

```
      //Use SIMD to initialize Distance while states are in sets of four
Init4Dist:
      movq   mm0, [ebx+edx]      ; mm0    = [pi3     pi2     pi1     pi0    ]
      paddsw mm0, [esi+edx]      ; mm0    = [pi3+b03 pi2+b02 pi1+b01 pi0+b00]
      movq   [edi+edx], mm0   ; dist[] = mm0
      add    edx, 8
      jnz    Init4Dist

      cmp     ecx, 0
      jz    InitDone

      // edx equals zero before entering loop
      // Initialize Distance of remaining states, states not in a set of four
      // Employed MMX(tm) technology in order to use the add with saturate instruction
Init1Dist:
      pinsrw   mm0, [ebx+edx*2], 0  ; mm0     = pi[i]
      paddsw      mm0, [esi+edx*2]     ; mm0     = pi[i] + bProb[ObsVect[0]][i]
      pextrw   eax, mm0, 0          ; dist[i] = pi[i] + bProb[ObsVect[0]][i]
      mov         [edi+edx*2], ax
      inc         edx
      cmp         edx, ecx
      jl       Init1Dist

InitDone:
      movq   mm3, mask0       ; mm3 = [7FFF 7FFF 7FFF 7FFF]
      movq   mm4, mask1       ; mm4 = [7FFF 7FFF 7FFF 0   ]
      movq   mm5, mask2       ; mm5 = [7FFF 7FFF 0    0   ]
      movq   mm6, mask3       ; mm6 = [7FFF 0    0    0   ]

      mov  edi, distBuffer    ; edi = &distBuffer[0]
      xor     ecx, ecx

      //Loop for each observation or obsCount
mainLoop:
      mov   edx, set4States_x_2
      inc   ecx                 ; next observation
      cmp      ecx, obsLen
      je    mainLoopDone
      mov      esi, bProb        ; esi = &bProb[0]
      mov      eax, obsVect         ; eax = &obsVect[0]
      mov      eax, [eax+ecx*4]  ; eax = obsVect[ecx]
      mov      esi, [esi+eax*4]  ; esi = &bProb[obsVect[ecx]][0]
      mov      ebx, aProb
      xor   eax, eax

      //Loop to Find the Dist for each state per observation
      cmp    edx, 0
      je     CheckLastStates
```

```
FourStateLoop:
      movq    mm0,  [edi+eax]    ; mm0 = [d3     d2     d1     d0   ]
      movq    mm1,  [edi+eax+2]  ; mm1 = [d4     d3     d2     d1   ]
      paddsw  mm0,  [ebx+eax]    ; mm0 = [d3+a3  d2+a2  d1+a1  d0+a0]
      paddsw  mm1,  [ebx+eax+8]  ; mm1 = [d4+a7  d3+a6  d2+a5  d1+a4]
      movq    mm2,  [edi+eax+4]  ; mm2 = [d5     d4     d3     d2   ]
      paddsw  mm2,  [ebx+eax+16] ; mm2 = [d5+a11 d4+a10 d3+a9  d2+a8]
      pminsw     mm0,  mm1
      pminsw  mm0,  mm2          ; mm0 = min(Dist+aProb) = min(mm0,mm1,mm2)
      paddsw  mm0,  [esi+eax]    ; mm0 += bProb[obsVect[ecx]][i]
      movq  [edi+eax], mm0       ; store distance
      add      eax,  8
      add      ebx,  16
      cmp      eax,  edx
      jnz      FourStateLoop

CheckLastStates:
      mov      edx, nRemain
      cmp      edx, 1            ; if one state left, handle last State
      je    lastState
      cmp      edx, 2            ; if two states left, handle the last two states
      je    lastTwoStates
      cmp      edx, 3            ; if three states left, handle the three states
      je    lastThreeStates
      jmp      mainLoop       ; else get distances for next observation


lastThreeStates:
      // Three states left to process

      movq  mm2, [edi+eax+4]  ; mm2 = [xx   xx    xx    d2   ]
      movq  mm1, [edi+eax+2]  ; mm1 = [xx   xx    d2    d1   ]
      paddsw    mm2, [ebx+eax+16]   ; mm2 = [xx   xx    xx    d2+a8]
      pand  mm2, mm3          ; remove sign bit
      paddsw mm1, [ebx+eax+8] ; mm1 = [xx   xx    d2+a5 d1+a4]
      movq  mm0, [edi+eax]    ; mm0 = [xx   d2    d1    d0   ]
      paddsw mm0, [ebx+eax]   ; mm0 = [xx   d2+a2 d1+a1 d0+a0]

      por       mm2, mm4      ; mm2 = [7FFF 7FFF  7FFF  d2+a8]
      pminsw mm1, mm2         ; mm1 = [xx   xx    d2+a5 min  ]

      pand  mm1, mm3       ; remove sign bit
      por      mm1, mm5    ; mm1 = [7FFF 7FFF  d2+a5 min  ]
      pminsw   mm0, mm1    ; mm0 = [xx   d2+a2 min1  min0 ]

      paddsw      mm0, [esi+eax] ; mm0 += bProb[obsVect[ecx]][i]
      movq     [edi+eax], mm0    ; store distance
      jmp      mainLoop
```

```
lastTwoStates:
      // Two states left to process

      movd  mm1, [edi+eax+2]  ; mm1 = [xx    d1   ]
      paddsw mm1, [ebx+eax+8] ; mm1 = [xx    d1+a4]
      pand  mm1, mm3          ; remove sign bit

      movd  mm0, [edi+eax]    ; mm0 = [d1    d0   ]
      paddsw    mm0, [ebx+eax]; mm0 = [d1+a1 d0+a0]

      por         mm1, mm6    ; mm1 = [7FFF  d1+a4]
      pminsw   mm0, mm1       ; mm0 = [d1+a1 min  ]

      paddsw      mm0, [esi+eax] ; mm0 += bProb[obsVect[ecx]][i]
      movd     [edi+eax], mm0    ; store distance
      jmp         mainLoop

lastState:
      // Last state to process
      // use MMX(tm) technology in order to take advantage of the paddsw instruction

      pinsrw      mm0, [edi+eax],0  ; mm0 = [xx  xx  xx  d0   ]
      paddsw      mm0, [ebx+eax]    ; mm0 = [xx  xx  xx  d0+a0]
      paddsw      mm0, [esi+eax]    ; mm0 += bProb[obsVect[ecx]][i]
      pextrw      edx, mm0,0        ; store distance
      mov      [edi+eax], dx     ; store distance
      jmp         mainLoop

      // Get Minimum Distance
mainLoopDone:
      mov     ecx, nStates
      cmp        ecx, 8
      jg      SIMD4           ; Use SIMD if enough states are used in HMM model

      // If Less than nine states find the minimum distance w/o employing SIMD
      pinsrw   mm0, [edi], 0  ; mm0 = [xx xx xx m0]
      cmp      ecx, 1
      je    storeMin
      shl      ecx, 1
      add      edi, ecx       ; edi = &dist[nStates-1]
      neg      ecx               ; [edi+ecx] holds next distance
loopMin:
      add        ecx, 2          ; increment pointer to load next distance
      pinsrw  mm1, [edi+ecx],0   ; load next distance
      pminsw     mm0, mm1        ; place minimum distance in mm0
      jnz        loopMin

storeMin:
      pextrw  eax, mm0, 0        ; store minimum distance and exit
```

```
      mov     minDist, ax
      jmp        Exit

      // If nStates > 4, use SIMD to find minimum distance
SIMD4:
      mov      ecx, SIMD4MinCount   ; ecx = num of states in sets of four
      add      edi, ecx             ; edi = &dist[SIMD4MinCount-1]
      neg      ecx                  ; [edi+ecx] will hold next four distances
      movq  mm0, [edi+ecx]          ; load first four distances
      add      ecx, 8               ; increment pointer to load next four distances

loopMinSIMD4:
      movq  mm1, [edi+ecx]          ; load next four distances
      pminsw mm0, mm1               ; place minimum four distances in mm0
      add      ecx, 8               ; increment pointer to load next four distances
      jnz   loopMinSIMD4

      // load the last four distances, store the minimum four distances
      mov      edi, lastFour     ; edi = &dist[nstates-5]
      movq    mm1, [edi]         ; load last four distances
      pminsw  mm0, mm1           ; place minimum four distances in mm0

      // Get the minimum distance from the four minimum distances
                              ; mm0 = [m3 m2 m1   m0  ]
      pshufw  mm1, mm0,0x0E   ; mm1 = [xx xx m3   m2  ]
      pminsw  mm0, mm1        ; mm0 = [xx xx min2 min1]
      pshufw  mm1, mm0,0x01   ; mm1 = [xx xx xx   min2]
      pminsw  mm0, mm1        ; mm0 = [xx xx xx   min ]
      pextrw   eax, mm0,0     ; eax = xx   min
      mov      minDist, ax    ; store min
      }
Exit:
      _asm {emms}
   return(minDist);
   }
```